ICOT Technical Memorandum: TM-0632

TM-0632

Extended Projection Method in Proofs-as-Programs

by
Y. Takayama

December, 1988

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Extended Projection Method in Proofs-as-Programs (Extended Abstract)

Yukihide Takayama

Institute for New Generation Computer Technology
1-4-28, Mita, Minato-ku, Tokyo, 108, Japan
takayama@icot.jp

subject: development of a mathematical method of program synthesis

1. Introduction

Writing programs as (constructive) proofs of theorems is a good approach to automated programming and program verification. Extracting an executable code, which is called a *realiser*, from a constructive proof by using the notion of realisability, or, equivalently, the Curry-Howard isomorphism [Howard 80], is one of the ways to make proofs run on computers. This raises the problem of extracting efficient codes from proofs. As pointed out in [Bates 79], the codes extracted by naive application of realisability contain a lot of inefficiency. Various techniques to eliminate the inefficiency have been developed so far. Source-to-source transformation rules to simplify realiser codes, proof normalisation and the simplification technique of the decision procedure are widely used in most of the implementations of constructive logic. Also, the pruning technique to remove redundant decision procedures given in [Goad 80], and code simplification for Harrop formulae such as the singleton justification in [Sasaki 86] and the rank 0 formulae in PX [Hayashi 88] have been developed. All these techniques are becoming almost standard.
However, realiser codes are still inefficient. For example, let $\forall x.\exists y.A(x,y)$ be a specification of of a function with the input, x, and the output, y. The function, f, which satisfies $\forall x.A(x,f(x))$ is extracted from the proof, and it is the only code that needs to be extracted in most situations. However, the code which is the computational meaning of the proof of $A(x,f(x))$ is also extracted, and the code is redundant. The Göteborg group approaches this problem by introducing the set type, $\{x : A|B\}$, to the Martin-Löf theory of types [Nordström 83], and the Nuprl group also uses the set type and the squash operator [Constable 86]. [Paulin-Mohring 87] introduced two constants, *Type* and *Spec*, in the calculus of construction [Coquand 88], and the class of the rank 0 formulae in PX also contains $\Diamond$-bounded formulae. These notations are introduced to declare which part of the proof is unnecessary in program extraction.
This paper presents another metamathematical approach, the *extended projection method*, to eliminate the redundancy. The underlying constructive logic and the style of specification and proof description are not changed; instead, the *declaration* and *marking* are introduced as the tools for performing proof tree analysis, which can be seen as a sort of program analysis in the context of proofs-as-programs. The redundancy in the constructive proof can be detected and eliminated automatically by giving a simple declaration to the specification. Also, several kinds of programs can be extracted from the same proof just by changing the declaration. The crucial

part of this method is handling the proofs in the induction rule. This paper also gives the proof theoretic background of the induction proof case.

2. A Simple Constructive Logic: QPC_0

The constructive logic used in this paper is called QPC_0, which is basically an intuitionistic first order natural deduction system with mathematical induction, higher order equalities and inequalities of terms, and primitive types, *nat* (natural number) and *bool*. It is a sugared subset of QJ [Sato 86].

A sort of q-realisability is given to QJ. The unique features of the realisability are as follows:

(1) Every realiser code is expressed as a sequence of terms;

(2) The realiser code for an atomic formula (or a Harrop formula) is a nil sequence (like Hayashi's *px*-realiser [Hayashi 88]);

(3) The realiser code extracted from an induction proof is defined as the solution of the system of fixed point equations. Generally, it is a multi-valued recursive call function.

The algorithmic realisability called the *Ext* procedure is given to QPC_0 [Takayama 88].

3. Declaration and Marking of Proof Trees

3.1 Declaration

The realiser code generated by *Ext* denotes the $\exists$-$\vee$ information in the specification, i.e., the value t which satisfies $A(t)$ for the formula, $\exists x.A(x)$, and the constant, $left$ or $right$, that indicates which formula, A or B, holds in the formula $A \vee B$.

The declaration defined below indicates which $\exists$-$\vee$ information of a given theorem is needed. It is the only information that end users of the system need to specify; the other part can be performed automatically.

Definition 1: Realising variable sequence of A, $Rv(A)$

1. $Rv(A) \stackrel{\text{def}}{=} ()$, if A is atomic;
2. $Rv(A \wedge B) \stackrel{\text{def}}{=} (Rv(A), Rv(B))$ Concatenation of the two sequences,$Rv(A)$ and $Rv(B)$;
3. $Rv(A \vee B) \stackrel{\text{def}}{=} (z, Rv(A), Rv(B))$ where z is a new variable;
4. $Rv(A \supset B) \stackrel{\text{def}}{=} Rv(B)$;
5. $Rv(\forall x : Type.\ A(x)) \stackrel{\text{def}}{=} Rv(A(x))$.
6. $Rv(\exists x : Type.\ A(x)) \stackrel{\text{def}}{=} (z, Rv(A(x)))$ where z is a new variable;

where $(a, b, \cdots, c)$ denotes a sequence of terms and () is the nil sequence.

Definition 2: Length of formulae

$l(A)$, which is called the *length of formula* A, is the length of $Rv(A)$.

Definition 3: Declaration

(1) A *declaration* of a specification, A, is the finite set, I, of offsets of $Rv(A)$. It is a subset of $\{0, 1, \cdots, l(A) - 1\}$. A specification, A, with the declaration, I, is denoted $\{A\}_I$. The elements of a declaration are called *marking numbers*.

(2) The empty set, ϕ, is called a *nil declaration.*

(3) The declaration, $\{0, 1, \cdots, l(A)-1\}$, is called *trivial.*

Example:

Let $A \stackrel{\text{def}}{=} \forall x.\ (x \geq 3 \supset \forall y.\exists z.\exists w.\ x = y \cdot z + w)$. $Rv(A) = \{z_0, z_1\}$, where z_0 corresponds to $\exists z$ and z_1 to $\exists w$. If the function that calculates the value of $\exists w$ from x is needed, the declaration of A is $\{1\}$.

3.2 Marking

If a declaration is given to the specification, the information can be inherited from bottom to top of the proof tree being reformed according to the inference rule of each application. Therefore, the same kind of information to declaration can be set to every node of the proof tree by using the declaration as the initial value. The information attached to each node is called *marking*, and the algorithm to calculate it is called $Mark$. Note that the declaration can be seen as a special case of marking. The *marked proof tree* is a tree obtained from a proof tree and the declaration by $Mark$.

The Ext procedure can use the information to refrain from generating unnecessary code. This is called *extended projection.*

The following set operations are used in $Mark$:

$I + n \stackrel{\text{def}}{=} \{x + n \mid x + n \leq max(I), x \in I\}$ $\qquad$ $I - n \stackrel{\text{def}}{=} \{x - n \mid x - n \geq 0, x \in I\}$

The following is part of the definition of the $Mark$ procedure:

(1) $Mark$ for the ($\exists$-I) rule

$$Mark\left(\dfrac{\dfrac{\Sigma}{t \quad A(t)}}{\{\exists x.\ A(x)\}_I}(\exists\text{-}I)\right) \stackrel{\text{def}}{=} \begin{cases} \dfrac{\{t\}_\phi \qquad Mark\left(\dfrac{\Sigma}{\{A(t)\}_{I-1}}\right)}{\{\exists x.\ A(x)\}_I}(\exists\text{-}I) & \text{if } 0 \notin I\ ; \\ \dfrac{\{t\}_{\{0\}} \qquad Mark\left(\dfrac{\Sigma}{\{A(t)\}_{I-1}}\right)}{\{\exists x.\ A(x)\}_I}(\exists\text{-}I) & \text{if } 0 \in I. \end{cases}$$

(2) $Mark$ for the ($\exists$-E) rule

$$Mark\left(\dfrac{\dfrac{\Sigma_0}{\exists x.\ A(x)} \qquad \dfrac{[t,\ A(t)]}{\dfrac{\Sigma_1}{C}}}{\{C\}_I}(\exists\text{-}E)\right) \stackrel{\text{def}}{=} \dfrac{Mark\left(\dfrac{\Sigma_0}{\{\exists x.\ A(x)\}_K}\right) \qquad Mark\left(\dfrac{[t,\ A(t)]}{\dfrac{\Sigma_1}{\{C\}_I}}\right)}{\{C\}_I}(\exists\text{-}E)$$

where

$$K = \begin{cases} M + 1 & \text{if } L = \phi \\ \{0\} \cup (M+1) & \text{if } L = \{0\} \end{cases}$$

and L and M are the unions of the markings of all the occurrences of t and $A(t)$ as hypotheses obtained in $Mark([t, A(t)]/\Sigma_1/\{C\}_I)$.

(3) $Mark$ for the ($\vee$-E) rule

$$Mark\left(\dfrac{\dfrac{\Sigma_0}{A\vee B}\quad\dfrac{\overset{[A]}{\Sigma_1}}{C}\quad\dfrac{\overset{[B]}{\Sigma_2}}{C}}{\{C\}_I}(\vee\text{-}E)\right)$$

$$\overset{\text{def}}{=}\begin{cases}\dfrac{Mark\left(\dfrac{\Sigma_0}{\{A\vee B\}_K}\right)\quad Mark\left(\dfrac{\overset{[A]}{\Sigma_1}}{\{C\}_I}\right)\quad Mark\left(\dfrac{\overset{[B]}{\Sigma_2}}{\{C\}_I}\right)}{\{C\}_I}(\vee\text{-}E) & \cdots \text{if } I\neq\phi\\ \text{all the nodes in the subtrees are marked } \phi & \cdots \text{otherwise}\end{cases}$$

where $K=\{0\}\cup(J_0+1)\cup(J_1+1+l(A))$, and J_0 and J_1 are the unions of the markings of all the occurrences of A and B as hypotheses.

(4) $Mark$ for the ($\supset$-E) rule

$$Mark\left(\dfrac{\dfrac{\Sigma_0}{A}\quad\dfrac{\Sigma_1}{A\supset B}}{\{B\}_I}(\supset\text{-}E)\right)\overset{\text{def}}{=}\dfrac{\dfrac{\Sigma_0}{A}\quad Mark\left(\dfrac{\Sigma_1}{\{A\supset B\}_I}\right)}{\{B\}_I}(\supset\text{-}E)$$

Every node in (Σ_0/A) has trivial marking.

4. Extended Projection Method Applied to Induction Proofs

4.1 Code from Induction Proofs

Ext for the mathematical induction is defined as follows:

$$Ext\left(\dfrac{\dfrac{\Sigma_0}{A(0)}\quad\dfrac{\overset{[x:nat,A(x)]}{\Sigma_1}}{A(x+1)}}{\forall x:nat.\ A(x)}(nat\text{-}ind)\right)\overset{\text{def}}{=}\mu\ \overline{z}.\ \lambda\ x.\ \ if\ x=0\ then\ Ext\left(\dfrac{\Sigma_0}{A(0)}\right)\ else\ Ext\left(\dfrac{\overset{[x:nat,A(x)]}{\Sigma_1}}{A(x+1)}\right)\sigma$$

where $\overline{z}=Rv(A(x))$, and $\sigma=\{\overline{z}/\overline{z}(pred(x)),x/pred(x)\}$.

This means that the program extracted from an induction proof is, in general, a multi-valued recursive call function which calculates a sequence of length n ($=l(A(x))$) by using the sequence of the same length which is the realiser for the induction hypotheses. Then, how does the marking procedure work on an induction proof? The length of the sequence calculated by the recursive call function will be restricted by the declaration. Then, can the parameters of the fixed point operator, $\overline{z}$, be restricted in the same way? For example, assume that the length of $A(x)$ is n ($3<n$), $z_0,\cdots z_{n-1}\overset{\text{def}}{=}Rv(A(x))$ and the declaration, $\{0,1\}$, is given. Can the extracted code always be like $\mu(z_0,z_1).if\ x=0\ then\ \cdots$? The answer is negative because there may be, for example, an induction proof from which the following program is extracted:

$$\mu(z_0,z_1,z_2,z_3).\lambda x.\ if\ x=0\ then\ (t_0,t_1,t_2,t_3)\ else\ (F_0(z_1,x),F_1(z_1,z_2,x),F_2(z_2,x),F_3(z_3,x))$$

This code can be expanded into f_is which denote the values for z_is:

$f_0 \stackrel{\text{def}}{=} \mu z_0.\lambda x.if\ x = 0\ then\ t_0\ else\ F_0(f_1, x)$

$f_1 \stackrel{\text{def}}{=} \mu z_1.\lambda x.if\ x = 0\ then\ t_1\ else\ F_1(f_1, f_2, x)$

$f_2 \stackrel{\text{def}}{=} \mu z_2.\lambda x.if\ x = 0\ then\ t_2\ else\ F_2(f_2, x)$

$f_3 \stackrel{\text{def}}{=} \mu z_3.\lambda x.if\ x = 0\ then\ t_0\ else\ F_3(f_3, x)$

If the declaration for the specification were $\{0, 1\}$, the system would try to extract f_0 and f_1. However, f_2 occurs in these functions, and this means that f_2 is also necessary to calculate f_0 and f_1. Therefore, just taking z_0 and z_1 as the parameters of the fixed point operator does not work well, and it proves that restricting the extracted code to a sequence of length 2 is impossible in this case.

The phenomena explained above can be checked by the marking procedure. The marking procedure traces which $\exists$-$\forall$ information is really used to prove the conclusion. $Mark$ will give the marking, $\{1, 2\}$, to the induction hypothesis, which means that the first and second codes, f_1 and f_2, must be calculated at the recursive call step. Therefore, the declaration, $\{0, 1\}$, turns out to be too small, and should be enlarged to $\{0, 1, 2\}$.

Definition 4:

Let I be the declaration given to the conclusion of an induction proof, and J be the marking of the induction hypothesis given by $Mark$.

(1) If there is a marking number, i, such that $i \in I$ and $i \notin J$, then i is called *missed*, or a *missing marking number*;

(2) If there is a marking number, j, such that $j \in J$ and $j \notin I$, then j is called *overflowed marking number*.

4.2 Form of Normalised Induction Proofs

The interest here lies in investigating the form of proofs and the markings which cause missing and overflowed marking numbers. It is characterised in the class of the normalised proofs which have a structure that is easy to handle. However, the form of the normalised proof tree can be understood a little more specifically when the normalisation theory for intuitionistic natural deduction [Prawitz 65] is applied to an induction step proof.

Definition 5: Symmetric path & vertical proofs

Let Π be a normalised induction step proof: $A(x) \vdash A(x + 1)$.

(1) The path, π, from an occurrence of $A(x)$, which satisfies the following condition is, if it exists, called a *symmetric path*;

Condition: if there is a segment of the formula, $B(x)$, in the E-part of π with γ as its principal sign, and if it is a premise of an application of the (γ-E) rule, then there is a segment of the formula, $B(x + 1)$, in the I-part of π and it is a premise of an application of the (γ-I) rule.

(2) Π is called a *vertical proof* iff all the main paths from any occurrence of $A(x)$ are symmetric.

4.3 Proof Theoretic Characterisation of Missing Marking Numbers

The form of the marked proof trees that may cause the missing marking numbers is classified as follows:

(1) Critical ($\perp$-E) application $\cdots$ all the marking numbers are missed;

(2) Critical ($\wedge$-$I\&E$) marking $\cdots$ the marking numbers for some of the formulae which are connected by $\wedge$ are missed;

(3) Critical ($\exists$-$I\&E$) marking $\cdots$ the marking number for $\exists x$ in $\exists x.A(x)$ is missed;

(4) Critical ($\vee$-$I\&E$) marking $\cdots$ the marking numbers for some of the formulae which are connected by $\vee$ are missed.

Definition 6: Regular marked proof tree

Assume an induction step proof, Π, and let Π_m be the marked proof tree Π. Then, Π_m is called *regular* iff it has no critical ($\perp$-E) applications, and there are no critical markings along any of the symmetric paths, if they exist, from any occurrences of the induction hypothesis.

4.4 Proof Theoretic Characterisation of Overflowed Marking Numbers

The forms of the marked proof trees which may cause overflowed marking numbers are classified as follows:

(1) Critical ($\exists$-E) assumptions $\cdots$ when more than two kinds of $\exists$-information of the induction hypothesis are used to construct a term of $\exists$-information of the conclusion of the induction step proof;

(2) Critical ($\supset$-E) application $\cdots$ when one of the occurrences of the induction hypothesis is above a minor premise of an ($\supset$-E) application;

(3) Critical segments $\cdots$ when there is an application of ($\exists$-E) or ($\vee$-E) whose conclusion has a non-nil marking and the premise, $\exists x.A(x)$ or $A \vee B$, is on the deduction path from an occurrence of the induction hypothesis.

4.5 Marking Theorem

The missing and overflow marking phenomena vary according to the proofs and declarations. They occur as complex mixtures of critical applications, critical markings, critical assumptions and critical segments. However, the following theorem holds for regular vertical proofs:

Theorem 1:

Suppose that a formula, $\forall x.A(x)$, is proved by mathematical induction, and I is an arbitrary declaration to the conclusion. Let Π^{ind} be the vertical induction proof of the induction step, $A(x) \vdash A(x+1)$, and suppose that Π^{ind}_m, the marked version of Π^{ind}, is regular, then:

(1) If Π^{ind}_m has a critical ($\supset$-E) application in one of the paths from an occurrence of the induction hypothesis, $A(x)$, then the marking of $[A(x)]$ is trivial;

(2) If Π^{ind}_m has no critical ($\supset$-E) applications, critical segments, or critical ($\exists$-E) assumptions on any symmetric paths from any occurrences of the induction hypothesis, the marking of the induction hypothesis by Mark, $[A(x)]$, is I;

(3) If Π^{ind}_m has no critical ($\supset$-E) application but there are either critical segments or critical

($\exists$-E) assumptions on a path from an occurrence of $A(x)$ or both, the marking of $[A(x)]$ is a superset of I.

According to this theorem, the declaration of the conclusion should be as follows to construct the right recursive call functions from the vertical induction proofs.

Case 1: If the proof tree of the induction step has a critical ($\supset$-E) application in one of the main paths from the induction hypothesis, the declaration is trivial.

Case 2: If the proof tree of the induction step has no critical ($\supset$-E) applications, critical $\exists$-E assumptions, or critical segments, the declaration may be arbitrary.

Case 3: If the proof tree of the induction step has no critical ($\supset$-E) applications but has at least one critical segment or critical $\exists$-E assumption on one of the main paths from an occurrence of the induction hypothesis, there is a possibility that the declaration must be enlarged to eliminate critical segments. In this case, the marking of the induction hypothesis, S, and the initial declaration may be different, so that the declaration should be $S \cup U$ and perform the marking again. There is also a possibility that S contains some overflowed marking numbers when I is larger and $S \not\subset I$; however, I and S are bounded by the trivial marking, so that S becomes equal to I in finite steps of the above operation.

Note that the marking, overflow check, and re-marking cycle for vertical proofs can also be applied to non-vertical proofs.

5. Modified Proof Compilation Algorithm

Ext should be modified to handle marked proof trees. The chief modifications are:

1) If the given formula, A, is marked by $\{i_0, \cdots, i_k\}$, extract only the i_lth $(0 \leq l \leq k)$ realiser code from every subtree of the proof;

2) If the formula, A, is marked by ϕ, no code should be extracted;

3) If the formula, A, is trivially marked, apply the Ext procedure.

The modified Ext procedure will be called $NExt$ in the following description.

Theorem 2: $NExt$ procedure and projection

Let A be a sentence and D be the declaration. If $\vdash A$ and Π is its normalised proof tree, then

(1) If ($\supset$-I) is not used in Π, $NExt(Mark(\Pi)) = proj(D)(Ext(\Pi))$ (projection of $\forall i(\in D)$th elements);

(2) If A is the consequence of a ($\supset$-I) application and there is no other application of the rule in Π, then $NExt(Mark(\Pi))$ is equal to the code, T, that is obtained by the following procedure: a) Let $proj(D)(Ext(\Pi)) = \lambda\overline{x}.\ t_{\overline{x}}$, where $\overline{x}$ is the realising variables of the hypothesis of the ($\supset$-I) application; b) subtract the variables which do not occur in $t_{\overline{x}}$ from $\overline{x}$ to obtain a subsequence, $\overline{y}$; and c) let $T = \lambda\overline{y}.\ t_{\overline{x}}$.

6. Example

Here, example of a prime number checker program is investigated. The redundancy-free code is extracted by the extended projection method.

6.1 Extraction of a Prime Number Checker Program by Ext

The specification of the program which takes any natural number as input and returns the boolean value, T, when the given number is prime, otherwise returns F, is as follows:

Specification

$$\forall p : nat.\ \ (p \geq 2 \supset \exists b : bool.\ \ (\ (\forall d : nat.\ (1 < d < p \supset \neg(d \mid p)) \wedge b = T) \\ \vee (\exists d : nat.\ (1 < d < p \wedge (d \mid p)) \wedge b = F)))$$

where $(x \mid y) \stackrel{\text{def}}{=} \exists z.y = x \cdot z$.

This specification can be proved by using the following lemma which is proved by mathematical induction and two applications of ($\forall$-E) and an application of ($\forall$-I).

Lemma: $\forall p : nat.\ \forall z : nat.\ (z \geq 2 \supset A(p, z))$

where

$$A(p, z) \stackrel{\text{def}}{=} \exists b : nat.\ (P_0(p, z, b) \vee P_1(p, z, b))$$

$$P_0(p, z, b) \stackrel{\text{def}}{=} \forall d : nat.\ (1 < d < z \supset \neg(d \mid p)) \wedge b = T$$

$$P_1(p, z, b) \stackrel{\text{def}}{=} \exists d : nat.\ (1 < d < z \wedge (d \mid p)) \wedge b = F$$

The program extracted by Ext is as follows:

$$prime \stackrel{\text{def}}{=} \lambda p.\ Ext(Lemma)(p)(p)$$

$$Ext(Lemma) \stackrel{\text{def}}{=} \lambda p.\ \mu(z_0, z_1, z_2, z_3).$$
$$\lambda z.\ if\ z = 0\ then\ any[4]$$
$$else\ \ if\ z = 1\ then\ any[4]$$
$$else\ \ if\ z = 2\ then\ (T, left, any[2])$$
$$else\ \ if\ proj(0)((z_1, z_2, z_3)(z - 1)) = left \cdots (*)$$
$$then\ \ if\ proj(0)(PROP\sigma_0) = left$$
$$then\ (T, left, any[2])$$
$$else\ (F, right, z - 1,\ proj(1)(prop\sigma_0))$$
$$else\ (F, right, z_2(z - 1), z_3(z - 1))$$

$$PROP \stackrel{\text{def}}{=} if\ proj(1)Th(m, n) = 0\ then\ (right, proj(0)(Th(m, n)))$$
$$else\ (left, any[1])$$

where $\sigma_0 \stackrel{\text{def}}{=} \{m/p, n/z - 1\}$ and Th is the program extracted from the proof of the natural number division theorem. $Ext(Lemma)$ is a multi-valued recursive call function which calculates four sequences of terms. The boolean value which denotes whether the given number is prime is the first element of the sequence, so that the other part of the sequence seems to be redundant.

However, the decision procedure $(*)$ uses the second term of the sequence. This means that the second term of the sequence is also necessary. The other part, the third and fourth elements, is redundant.

6.2 Extraction of a Prime Number Checker by $NExt$

The meaning of the realising variable sequence, (z_0, z_1, z_2, z_3), of the specification is as follows: z_0 denotes $\exists$-information for $\exists b$; z_1 denotes $\vee$-information for $P_0(p,p,b) \vee P_1(p,p,b)$; z_2 denotes $\exists$-information for $\exists d$; z_3 denotes $\exists$-information for $(d \mid p) \stackrel{\text{def}}{=} \exists r : nat.\ p = r \cdot d$.
As the only information needed is whether the given natural number is prime or not, z_0 should be specified, i.e., the declaration is $\{0\}$. However, 1 turns out to be an overflowed marking number in the first application of $Mark$, then 1 is added to the initial marking and $Mark$ is performed again. Consequently, $NExt$ generates the following code:

$$
\begin{array}{l}
NExt(Lemma) \stackrel{\text{def}}{=} \lambda p.\ \mu(z_0, z_1). \\
\qquad \lambda z.\ if\ z = 0\ then\ any[2] \\
\qquad else\ \ if\ z = 1\ then\ any[2] \\
\qquad\quad else\ \ if\ z = 2\ then\ (T, left) \\
\qquad\qquad else\ \ if\ z_1(z-1) = left \\
\qquad\qquad\quad then\ \ if\ proj(0)(PROP\sigma_0) = left\ then\ (T, left) \\
\qquad\qquad\qquad\quad else\ (F, right) \\
\qquad\qquad\quad else\ (F, right)
\end{array}
$$

6.3 Extraction of Other Programs from the Same Proof

Another kind of program can be extracted from the same proof by changing the declaration to $\{1\}$. The extracted program is as follows:

$$prime_1 \stackrel{\text{def}}{=} \lambda p.T_1(p)(p)$$

$$
\begin{array}{l}
T_1 \stackrel{\text{def}}{=} \lambda p.\ \mu z_1. \\
\qquad \lambda z.\ if\ z = 0\ then\ any[1] \\
\qquad else\ \ if\ z = 1\ then\ any[1] \\
\qquad\quad else\ \ if\ z = 2\ then\ left \\
\qquad\qquad else\ \ if\ z_1(z-1) = left \\
\qquad\qquad\quad then\ \ if\ proj(0)(PROP\sigma_0) = left\ then\ left \\
\qquad\qquad\qquad\quad else\ right \\
\qquad\qquad\quad else\ right
\end{array}
$$

This is the program that returns $left$ if p is prime, and returns $right$ otherwise.
If the program which returns the minimum divisor of p when p is not prime is needed, it can be extracted by changing the declaration to $\{2\}$. Note that the overflow of the marking number happens, so that the program calculates 1st and 2nd elements.

7. Conclusion

A proof theoretic method to extract redundancy-free realiser code from a constructive logic was presented in this paper. The realiser codes of q-realisability contain some redundancy which can be seen as verification information. The redundancy can be removed by analysing of the length of formula occurrences in the given proof tree. The crucial part is the analysis of proofs in induction where the inference rules on logical constants are used in particular ways in the proof of induction step. These critical cases were specified from a proof theoretic point of view. The method presented in this paper automatically analyses and eliminates redundancy by making a simple declaration when the theorems and their proofs are set. The advantage of this method is that there is no need to change the underlying logic: the marking system, which is the additional information to proof trees, is independent of the base logic, QPC_0. Therefore, the method presented in this paper can be applied to other logic with minor modifications.

REFERENCES

[Bates 79] Bates, J.L., "*A logic for correct program development*", Ph.D. Thesis, Cornell University, 1979

[Constable 86] Constable, R.L., "*Implementing Mathematics with the Nuprl Proof Development System*", Prentice-Hall, 1986

[Coquand 88] Coquand, T. and Huet, G., "*The Calculus of Construction*", Information and Computation Vol. 76, 1988

[Goad 80] Goad, C.A., "*Computational Uses of the Manipulation of Formal Proofs*", Ph.D. Thesis, Stanford University, 1980

[Hayashi 88] Hayashi, S. and Nakano, H., "PX: A Computational Logic", The MIT Press, Cambridge, Massachusetts, 1988

[Howard 80] Howard, W. A., "The Formulae-as-types Notion of Construction", in *'Essays on Combinatory Logic, Lambda Calculus and Formalism'*, Eds. J. P. Seldin and J. R. Hindley, Academic Press, 1980

[Nordström 83] Nordström, B. and Petersson, K., "Types and specifications", Proceedings of IFIP'83, Elsevier, Amsterdam, 1983

[Paulin-Mohring 88] Paulin-Mohring, C., 1988, personal communication

[Prawitz 65] Prawitz, D., "*Natural Deduction*", Almqvist & Wiksell, 1965

[Sasaki 86] Sasaki, J., "*Extracting Efficient Code From Constructive Proofs*", Ph.D. Thesis, Cornell University, 1986

[Sato 86] Sato, M., "QJ: A Constructive Logical System with Types", France-Japan Artificial Intelligence and Computer Science Symposium 86, Tokyo, 1986

[Takayama 88] Takayama, Y., "**QPC**: QJ-Based Proof Compiler – Simple Examples and Analysis –", *European Symposium on Programming '88*, Nancy, 1988